Name Solution

# Computer Architecture
# EE 4720
# Midterm Examination

20 October 1999, 10:40–11:30 CDT

| | | |
|---|---|---|
| Problem 1 | | (33 pts) |
| Problem 2 | | (33 pts) |
| Problem 3 | | (34 pts) |
| Exam Total | | (100 pts) |

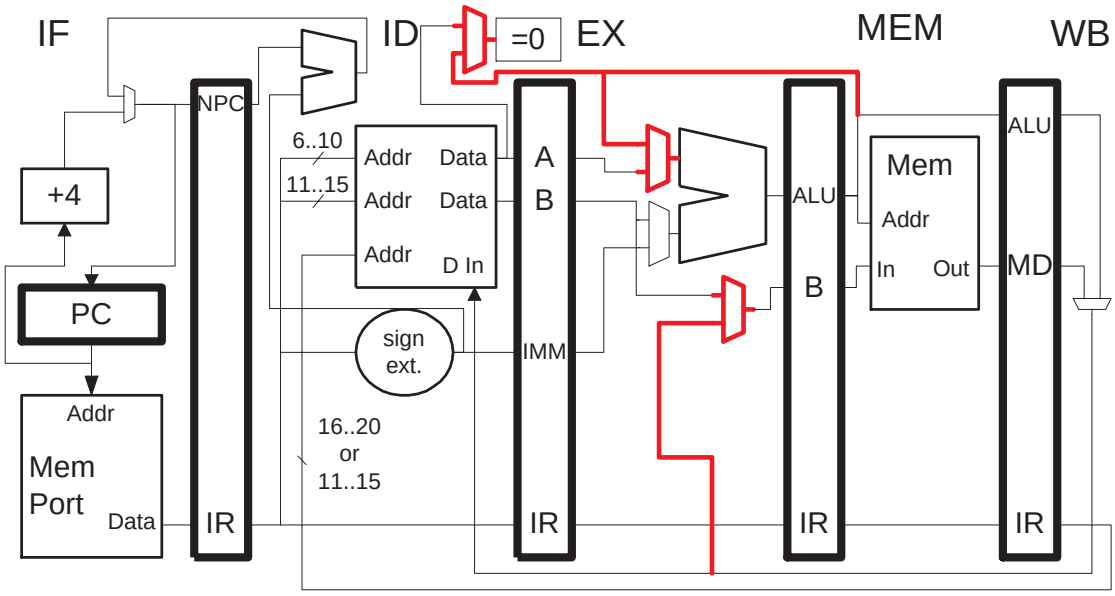
Alias

*Good Luck!*

Problem 1:

(*a*) Add *exactly* the bypass paths that are needed so the code below executes as shown in the pipeline execution diagram. (**Don't** add any bypass paths that are not needed by the code.) (15 pts)

Added bypass paths appear in **red bold** in the diagram below.



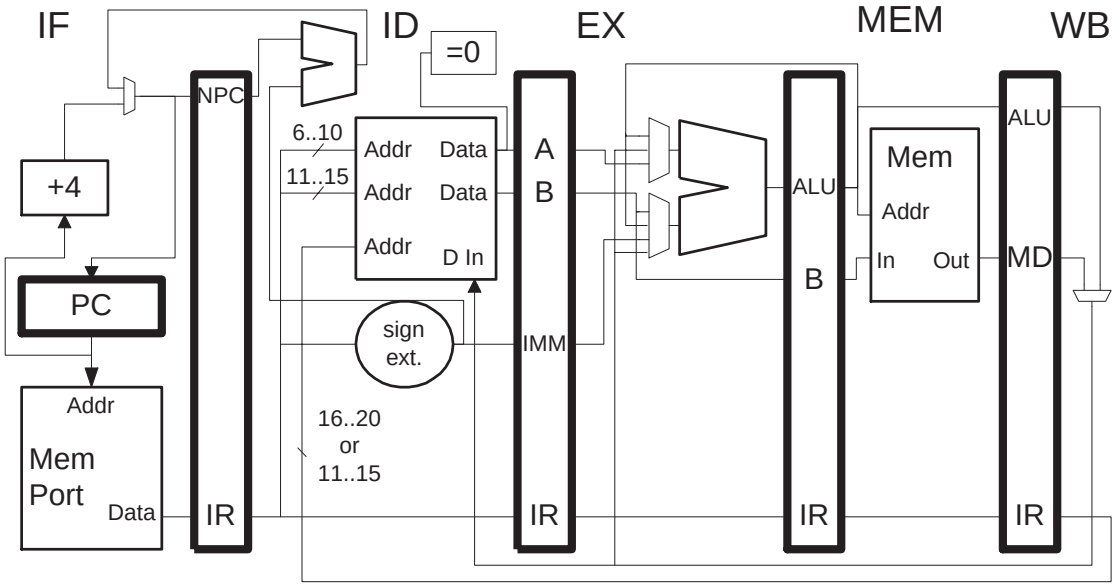

```
! r1 = 0x1010  r2 = 0x2020, r3 = 0x3030, r4 = 0x4040, r5 = 0x5050
! Cycle            0   1   2   3   4   5   6   7
LINE1:  LINE1 = 0x100
 lw   r1, 16(r2)   IF  ID  EX  MEM WB  IF  ID  EX ...
 addi r2, r2, #4       IF  ID  EX  MEM WB  IF  ID ...
 sw   20(r2), r1           IF  ID  EX  MEM WB  IF ...
 beqz r2, LINE1                IF  ID  EX  MEM WB ...
 and  r3, r4, r5               IFx
```

(*b*) Show the values of the registers listed below in **cycle 4** of the execution above using the added bypass paths. Assume that the load instruction retrieves a `0x1234` from memory. Instructions are squashed by replacing them with an `or r0, r0, r0`. (18 pts)

| IF.PC | ID/EX.A | EX/MEM.ALU | MEM/WB.ALU |
|---|---|---|---|
| 0X110 | 0X2020 | 0X2024 | 0X2030 |
| **IF/ID.NPC** | **ID/EX.B** | **EX/MEM.B** | **MEM/WB.MD** |
| 0X110 | 0X1010 | 0X2020 | 0X1234 |
| **IF/ID.IR** | **ID/EX.IMM** | **EX/MEM.IR** | **MEM/WB.IR** |
| beqz | 0X20 | addi | lw |
| | **ID/EX.IR** | | |
| | sw | | |

Problem 2:

(*a*) Show a pipeline execution diagram for the code below on the implementation shown until `lw` is fetched a second time. The first branch is not taken but the last one is. The only bypass paths available are the ones shown. (18 pts)




```
LOOP:
! Cycle          0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
 lw   r1, 0(r2)  IF  ID  EX  MEM WB                                          IF
 andi r3, r1, #4     IF  ID  --> EX  MEM WB
 beqz r3, LINE1          IF  --> ID  -----> EX  MEM WB
 add  r4, r4, r1             IF  -----> ID  EX  MEM WB
 j LINE2                                IF  ID  EX  MEM WB
LINE1:
 add  r5, r5, r1                            IFx
LINE2:
 sw   4(r2), r1                                 IF  ID  EX  MEM WB
 addi r2, r2, #8                                    IF  ID  EX  MEM WB
 bneq r2, LOOP                                          IF  ID  -----> EX
 xor  r5, r6, r7                                            IF  ----->x
! Cycle          0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
```

(*b*) Rewrite the code below (which is the same as the code on the previous page) using one-cycle delayed branches and predicated instructions and schedule the code so that it executes as quickly as possible. (Do not unroll the loop.) Assume that bypass paths are provided for the predicated instructions. It should be possible to remove all stalls, but if any remain point them out (for partial credit). (15 pts)

```
LOOP:
 lw   r1, 0(r2)
 andi r3, r1, #4
 beqz r3, LINE1
 add  r4, r4, r1
 j LINE2
LINE1:
 add  r5, r5, r1
LINE2:
 sw   4(r2), r1
 addi r2, r2, #8
 bneq r2, LOOP
 xor  r5, r6, r7
```

Predicated instructions are used in a straightforward manner.

To remove the stall between `lw` and `andi` the `addi` instruction is placed between them. Since `addi` was moved above `sw` eight is subtracted from the immediate in `sw`.

Now that branches have one-slot delays, `sw` can be placed after the branch.

Note that the code below encounters no stalls (assuming cache hits) and something useful is done in the branch delay slot.

```
LOOP:
       lw   r1, 0(r2)
       addi r2, r2, #8
       andi r3, r1, #4
 (~r3) add r4, r4, r1
 (r3)  add r5, r5, r1
       bneq r2, LOOP
       sw   -4(r2)
```

Problem 3: Answer each question below.

(*a*) Why do DLX branches (and branches in many other ISAs) use displacement addressing? Why don't branches use indirect addressing (destination address in a register) instead of displacement addressing? (8 pts)

The target of branches are usually nearby so the displacement (in the immediate field) in a type-J instruction would usually be large enough. If indirect addressing were used a register would have to be loaded with the target address, which would require two instructions before most branches.

(*b*) The code below executes on an implementation that uses a reservation register to detect WB structural hazards. At cycle zero the reservation register contains all zeros. Show the state of the reservation register at the end of each cycle below. Indicate which (if any) bit positions are tested in each cycle. (9 pts)

```
! Cycle              0  1  2  3  4  5  6  7  8  9  10
multf f0, f1, f2    IF ID M0 M1 M2 M3 M4 M5 WB
addf  f3, f4, f5       IF ID A0 A1 A2 A3 WB
subf  f6, f7, f8          IF ID -> A0 A1 A2 A3 WB
gtf   f9, f10, f11           IF -> ID A0 A1 A2 A3 WB
nop
nop
...
! Solution:
! Cycle            0  1  2  3  4  5  6  7  8  9  10
                Pos
                 7     1* 0  0  0  0  0  0  0  0  0
                 6     0  1  0  0  0  0  0  0  0  0
                 5     0  1* 1* 1* 1* 0  0  0  0  0
                 4     0  0  1  1  1  1  0  0  0  0
                 3     0  0  0  1  1  1  1  0  0  0
                 2     0  0  0  0  1  1  1  1  0  0

! Cycle            0  1  2  3  4  5  6  7  8  9  10
```

The contents of the reservation register is shown vertically each cycle except zero. An asterisk marks the bit position that is checked.

(*c*) Many packed operand instructions perform saturating arithmetic. What is saturating arithmetic? Provide an example. (8 pts)

In saturating arithmetic if the result of an operation is too large to represent (*e.g.*, 10 using 3 bits) it is replaced by the largest representable value (7 using 3 bit unsigned integers). Similarly if the result is too small, it is replaced by the smallest representable value.

Example: Eight-bit unsigned integers $100 + 200 = 255$.

(*d*) In homework 3, a special return address register (`ERA`) was used to hold exception return addresses. The jump and link instructions, `jal` and `jalr`, use `r31` for the return address; is this an option for exceptions? Explain. (9 pts)

It's not an option because `r31` must be left unchanged. After some exceptions (*e.g.*, page fault) execution is supposed to resume normally, so all register values must be left unchanged. The interrupted program might have been using `r31` for a jump and link return address, or for some other purpose.